ASSIGNMENT 2 - GROUP REPORT

BORCELLE

RESEARCH PROPOSAL

ARRANGED BY :
S3927539 - TO GIA HY
S3891909 - TRUONG BACH MINH
S3878071 - VO THANH THONG
S3957034 - TRUONG HONG VAN

Table of contents:

A.Introduction:

B.Application Design:

1) **Class diagram:**

   High-Level Design

   a) **Design:**

   b) **Relationship:**

   Low-Level Design:

   a) **User interface (UI):**

      a.1) Design choices:

      a.2)Components used:

   b) User interface (UI) Display:

2) **ERD :**
3) **Project limitations:**

C.Implementation:

A. Introduction

The project is built based on the e-commerce website of Lazada. An e-commerce website facilitates the purchase of goods by different sellers over the internet. The project requirements includes an admin account along with seller functions and customer functions as well. The project aims to not be a perfect recreation of the Lazada website but more of an interpretation based upon concepts introduced in this course. It is built upon the goal of making an efficient and user-friendly application to develop into a full-fledged product afterwards.

B. Application design

1. Class diagram

a. Design:

High-Level Design:

The application architecture has 3 tiers:

- Presentation layer: Reactjs
- Logic data layer : Nodejs, Expressjs
- Database layer: Mongodb

Components:

- User interface (UI).
- Authentication
- Product handling
- Orders
- Customer-Seller relationship

b. Relationship:

- The UI works with the authentication for access control
- The UI handle the product through product handling displaying different CRUDs elements
- The Customer-Seller relationship handles how the orders are supposed to display
- The product handling handles the product to create an order for the customer

Low-Level Design:

a. User interface (UI):

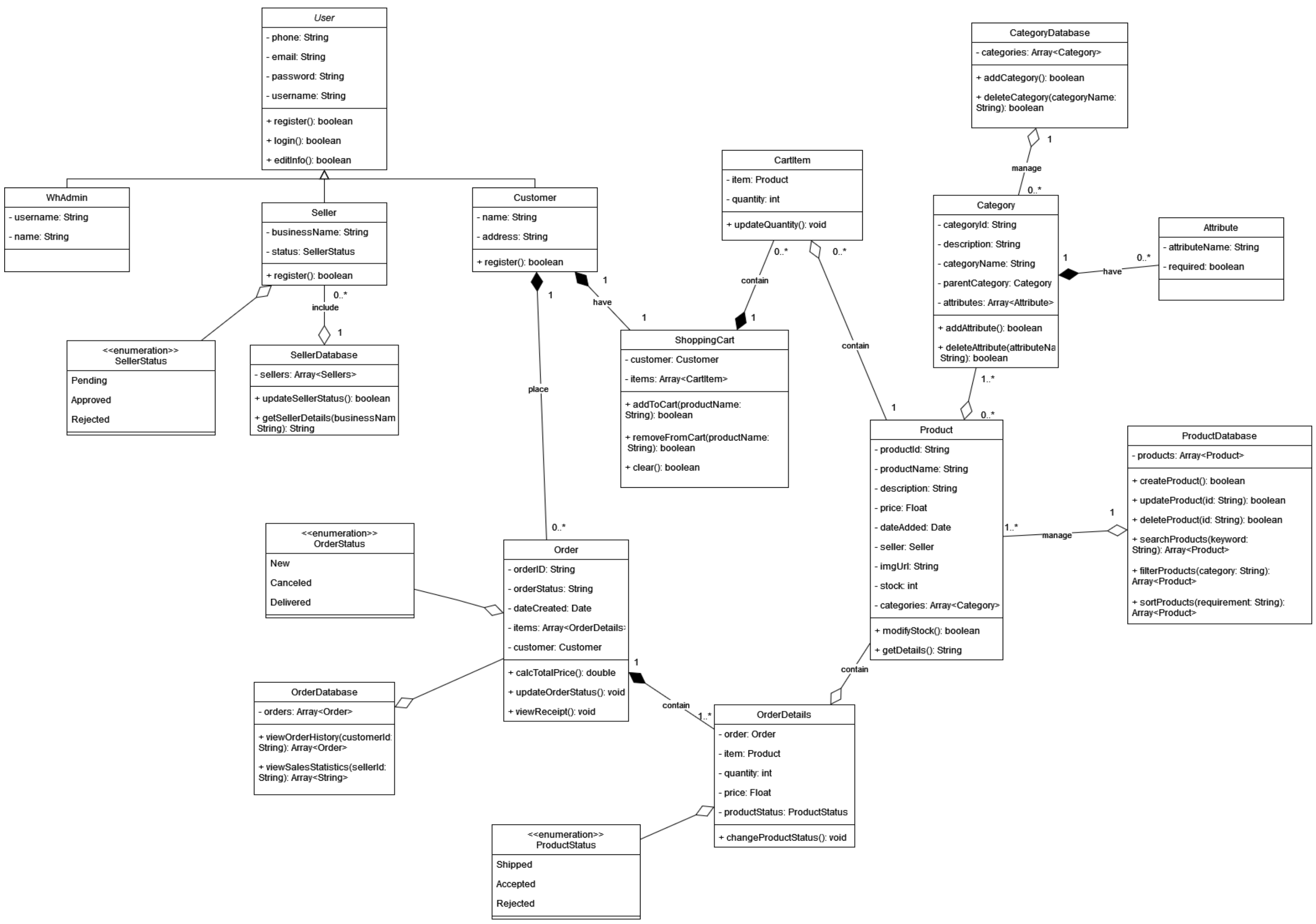
a.1. Design choices:

- ReactJS: Flexible and convenient to handle data from the database
- Bootstrap: Quick and provides many options to design the website

a.2. Components used:

- Homepage, Customer page, Seller page, Admin page

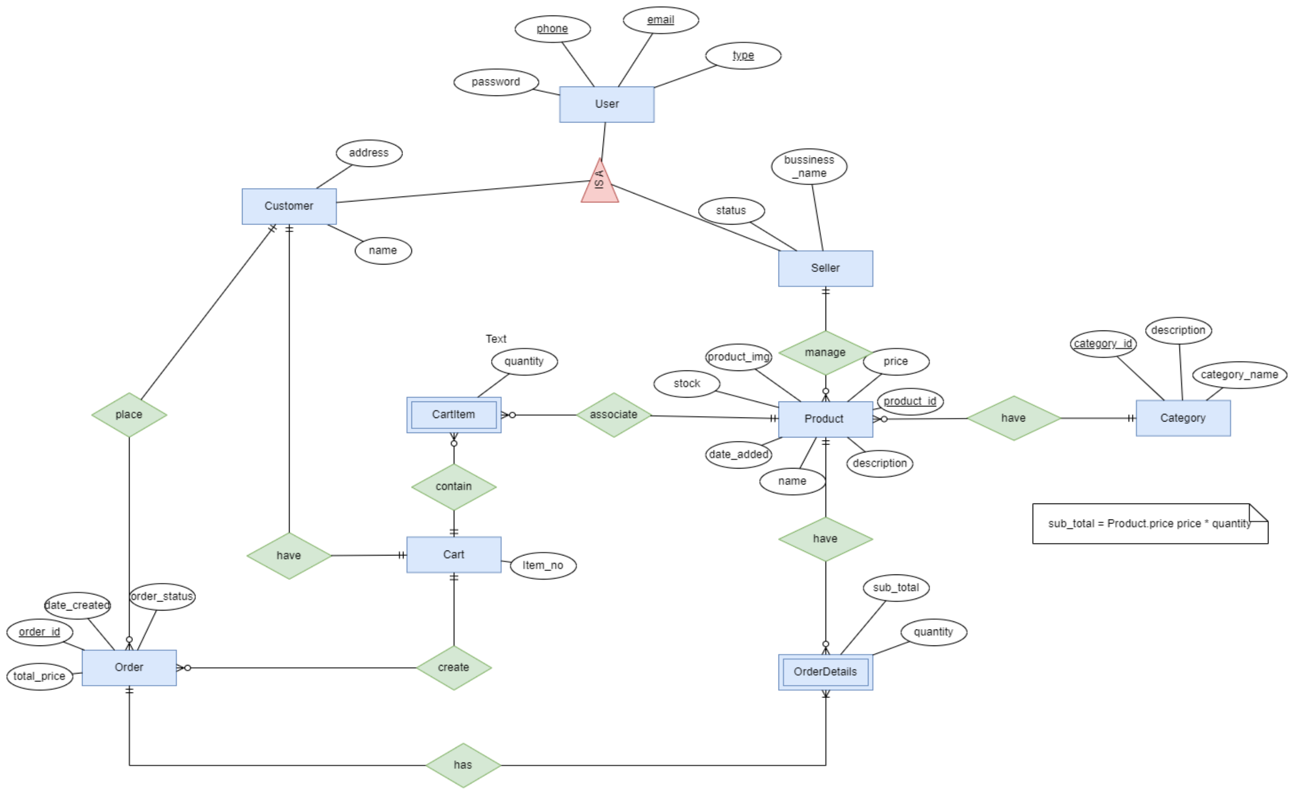
b. User interface (UI):



Here is a UML diagram to demonstrate the relationship between each database component. It was built to give an overview of what the database would look like before the development process. There are 3 types of user account: Admin, Seller and Customer. The Seller account has 3 statuses that can be set (Pending, Rejected and Approved). The information of the Seller is stored on the database application (Mongodb). The Customer account can add products to cart. Each product can have a category with many attributes but it is not necessary for a category to be linked with a product. The products are stored on the MongoDB database application. After adding to cart, the customer can place an order which is modifiable for the seller to change the order status based on if the order is Accepted, Rejected or already Shipped towards the customer. Each order placed will have its own detail. Which contains the products information, the quantity bought, the price and the Product status.

The Customer is entitled to a single cart. When they add a product, it is added into the cart. There is only one product database and all of the products are pulled from that database. The product List itself can still exist without any product hence the aggregation relationship. Each category has to include many attributes. The Order details pulls in the product's information to create itself. The product status is updated inside the Order by the Seller. There are 3 order statuses, it can be new, cancelled or delivered. The Seller account can have or not have the status. Therefore, it is linked with an aggregation relationship. The Seller Database shares the same relationship as it can have or not have any Seller account.

2. ERD



The Seller can manage zero to many products. The products can have one to many categories but the category can be not tied to any products created at the moment. The product can have zero to many order details as it is only included in the order details. The order details are only linked to a single order. The customer can place zero to many orders. A customer is linked to a single cart. A cart can contain zero to many cart items.

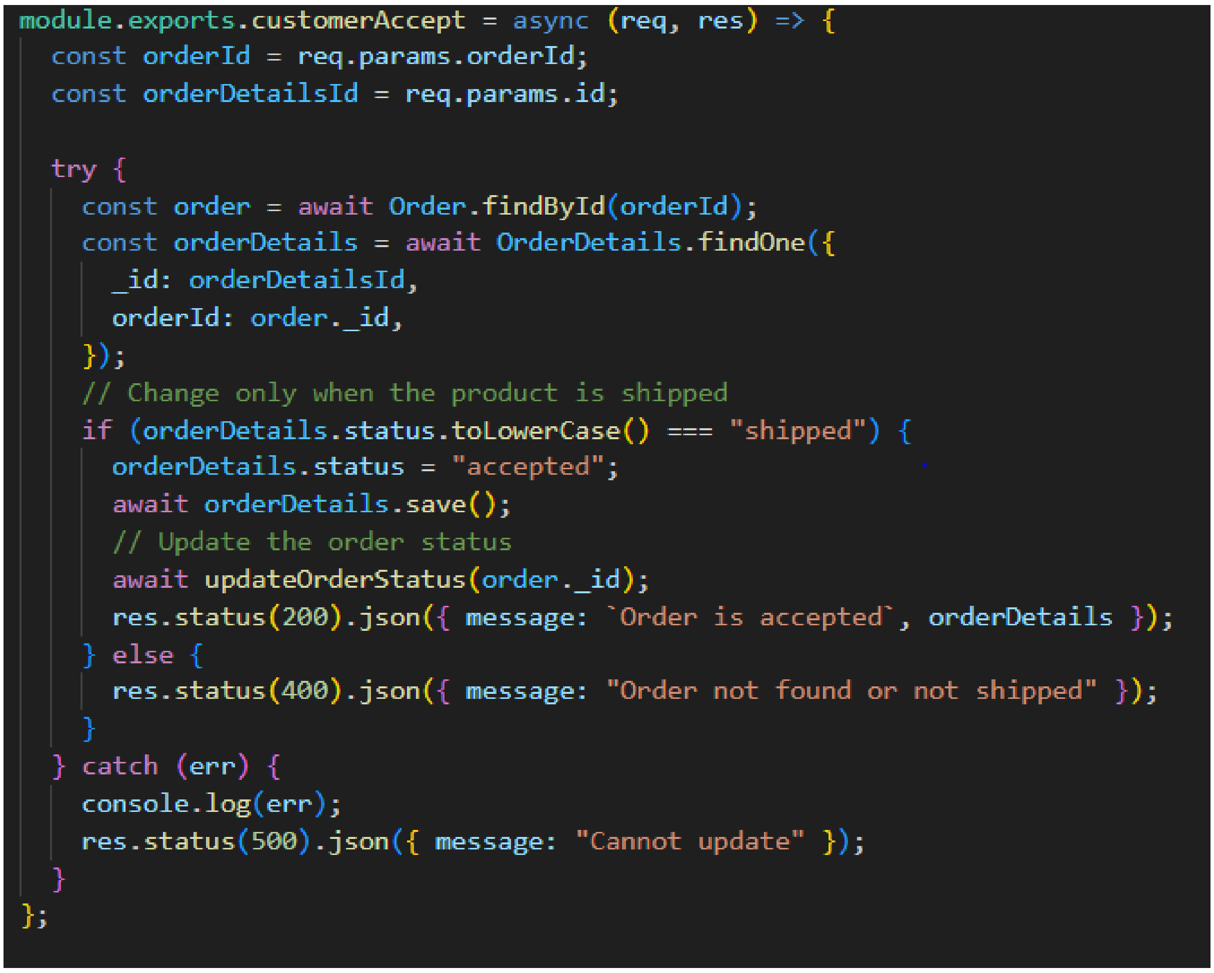
3. Project limitations:

During the development process, some limitations were identified for our project. Such as for the admin account, the admin cannot define the constraints for the attributes in a category. The Seller can enter the data freely. The seller also cannot edit any product's name. There is also a cart problem, we were able to save data from the customer to the local storage. However, the project requirements include the function of saving the cart data to the local storage when the user is not logged in. We find this to be a flawed security choice. Therefore, it was decided that we did not follow this requirement and implement it into the customer feature instead.

C. Implementation

I. Functional

a. Changing status (Seller to Order Status):

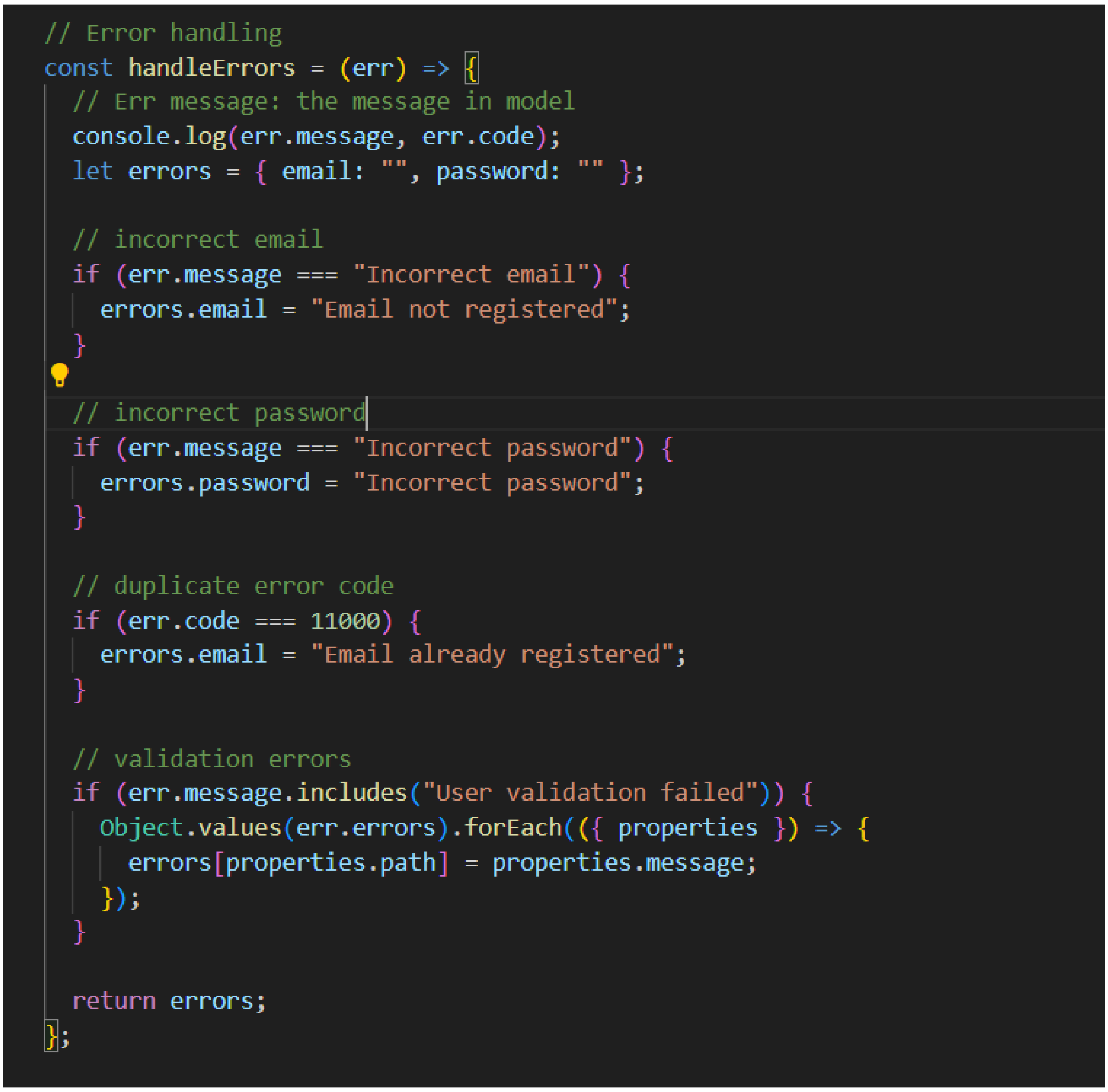


```
module.exports.customerAccept = async (req, res) => {
  const orderId = req.params.orderId;
  const orderDetailsId = req.params.id;

  try {
    const order = await Order.findById(orderId);
    const orderDetails = await OrderDetails.findOne({
      _id: orderDetailsId,
      orderId: order._id,
    });
    // Change only when the product is shipped
    if (orderDetails.status.toLowerCase() === "shipped") {
      orderDetails.status = "accepted";
      await orderDetails.save();
      // Update the order status
      await updateOrderStatus(order._id);
      res.status(200).json({ message: `Order is accepted`, orderDetails });
    } else {
      res.status(400).json({ message: "Order not found or not shipped" });
    }
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Cannot update" });
  }
};
```

The customerAccept function changes the status to accept when the Seller clicks on the accept button in the User interface. This will also be called when the customer pulls the api to see their order status. The other status change works the same way.

b. User authentication:

Error handling:



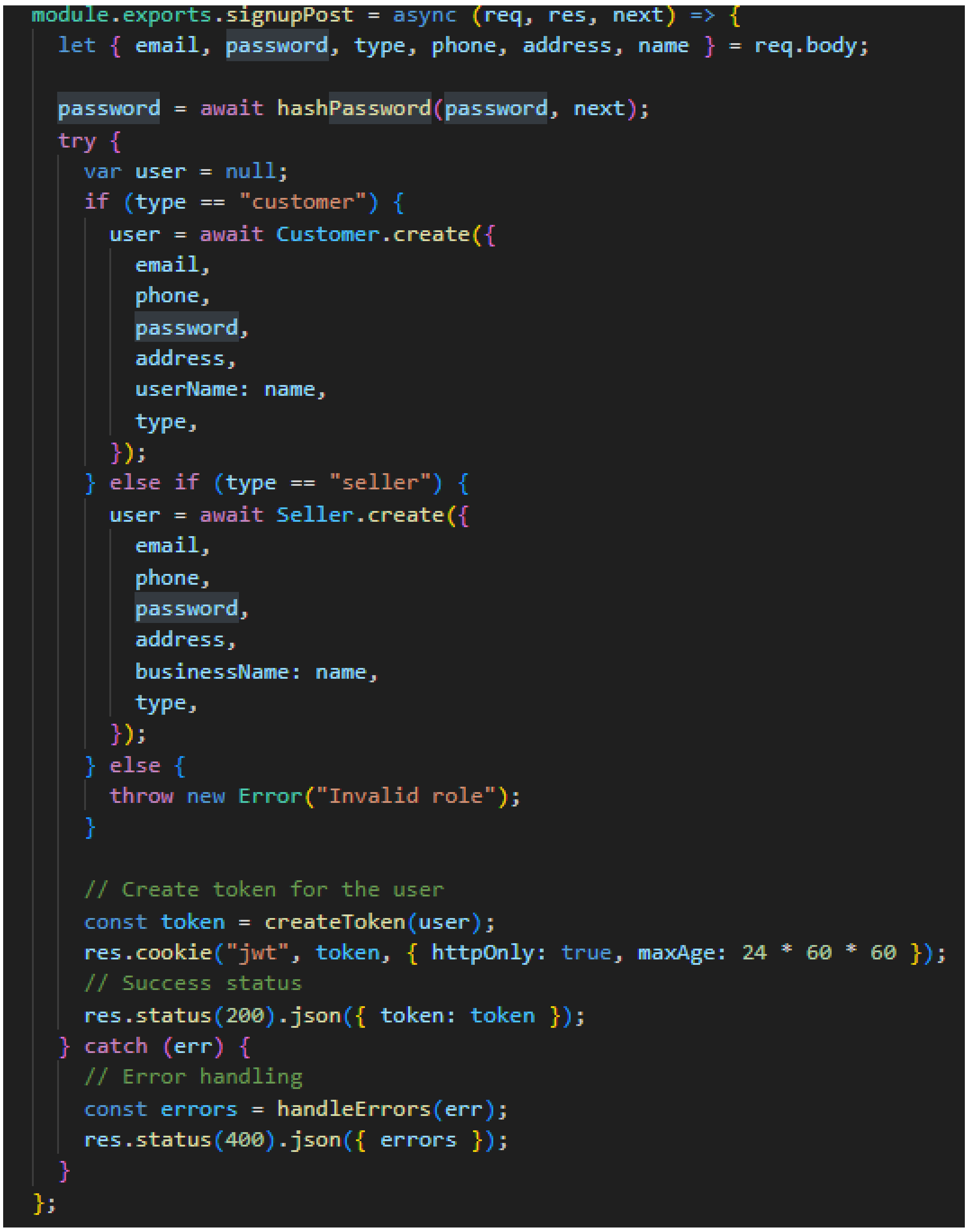
```
// Error handling
const handleErrors = (err) => {
  // Err message: the message in model
  console.log(err.message, err.code);
  let errors = { email: "", password: "" };

  // incorrect email
  if (err.message === "Incorrect email") {
    errors.email = "Email not registered";
  }

  // incorrect password
  if (err.message === "Incorrect password") {
    errors.password = "Incorrect password";
  }

  // duplicate error code
  if (err.code === 11000) {
    errors.email = "Email already registered";
  }

  // validation errors
  if (err.message.includes("User validation failed")) {
    Object.values(err.errors).forEach(({ properties }) => {
      errors[properties.path] = properties.message;
    });
  }

  return errors;
};
```

This is the error function to handle the error messages when the user enters the wrong username or password. If it is an incorrect email, then the function will respond with the "Email not registered" message. If it is an incorrect password, then the function responds with "Incorrect password".

Sign up:

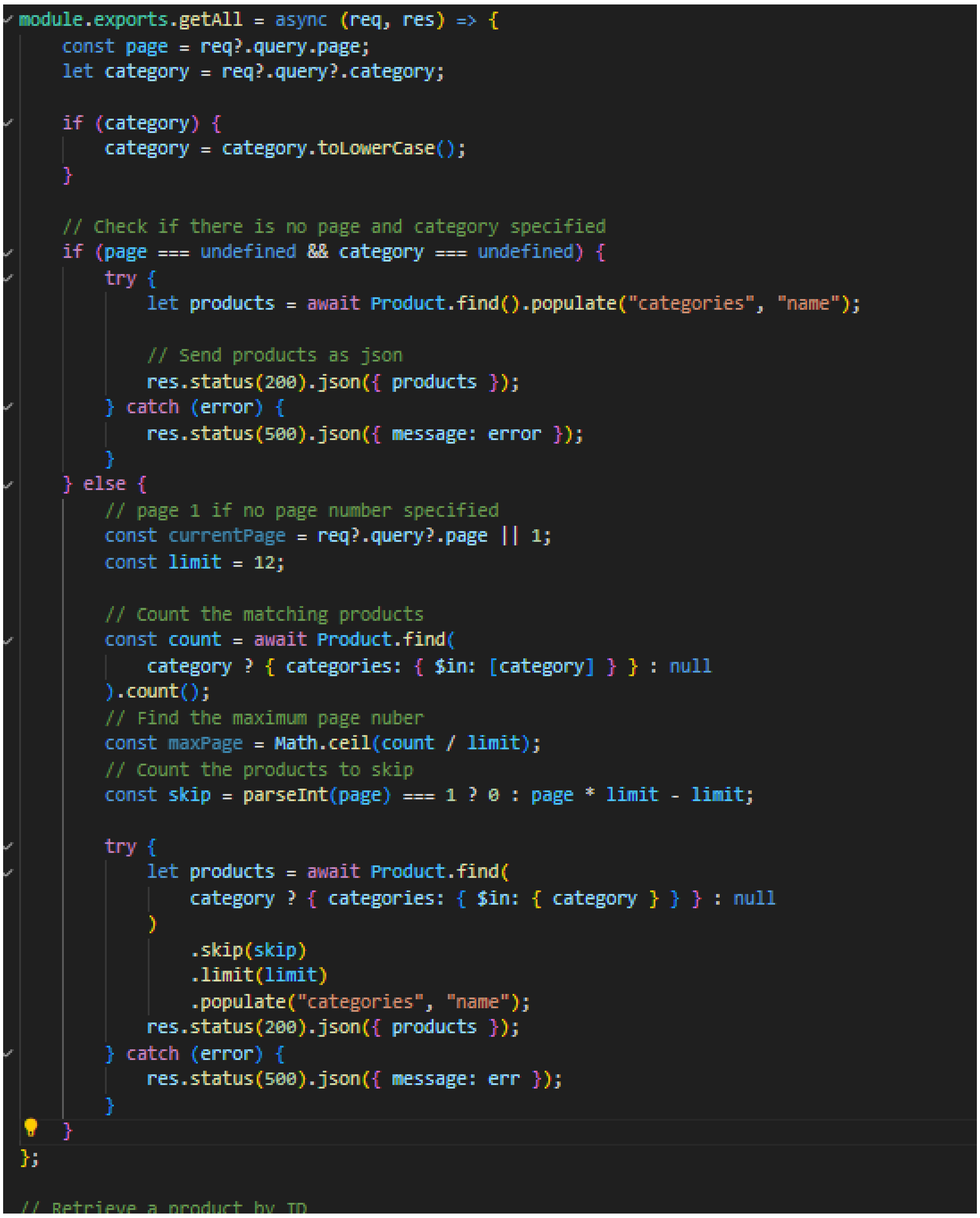
The sign up function creates the user account based on the if it is the Customer or the Seller. Then it creates a token for the account using jwt to authenticate the user when they get into the homepage.

The log in function checks the type of the account then verifies the username and the password with the correct database. If it is correct, then the token is generated to authenticate the user into the website. If it is wrong, then it throws an error message based on what the user entered wrong.



```javascript
module.exports.signupPost = async (req, res, next) => {
  let { email, password, type, phone, address, name } = req.body;

  password = await hashPassword(password, next);
  try {
    var user = null;
    if (type == "customer") {
      user = await Customer.create({
        email,
        phone,
        password,
        address,
        userName: name,
        type,
      });
    } else if (type == "seller") {
      user = await Seller.create({
        email,
        phone,
        password,
        address,
        businessName: name,
        type,
      });
    } else {
      throw new Error("Invalid role");
    }

    // Create token for the user
    const token = createToken(user);
    res.cookie("jwt", token, { httpOnly: true, maxAge: 24 * 60 * 60 });
    // Success status
    res.status(200).json({ token: token });
  } catch (err) {
    // Error handling
    const errors = handleErrors(err);
    res.status(400).json({ errors });
  }
};
```

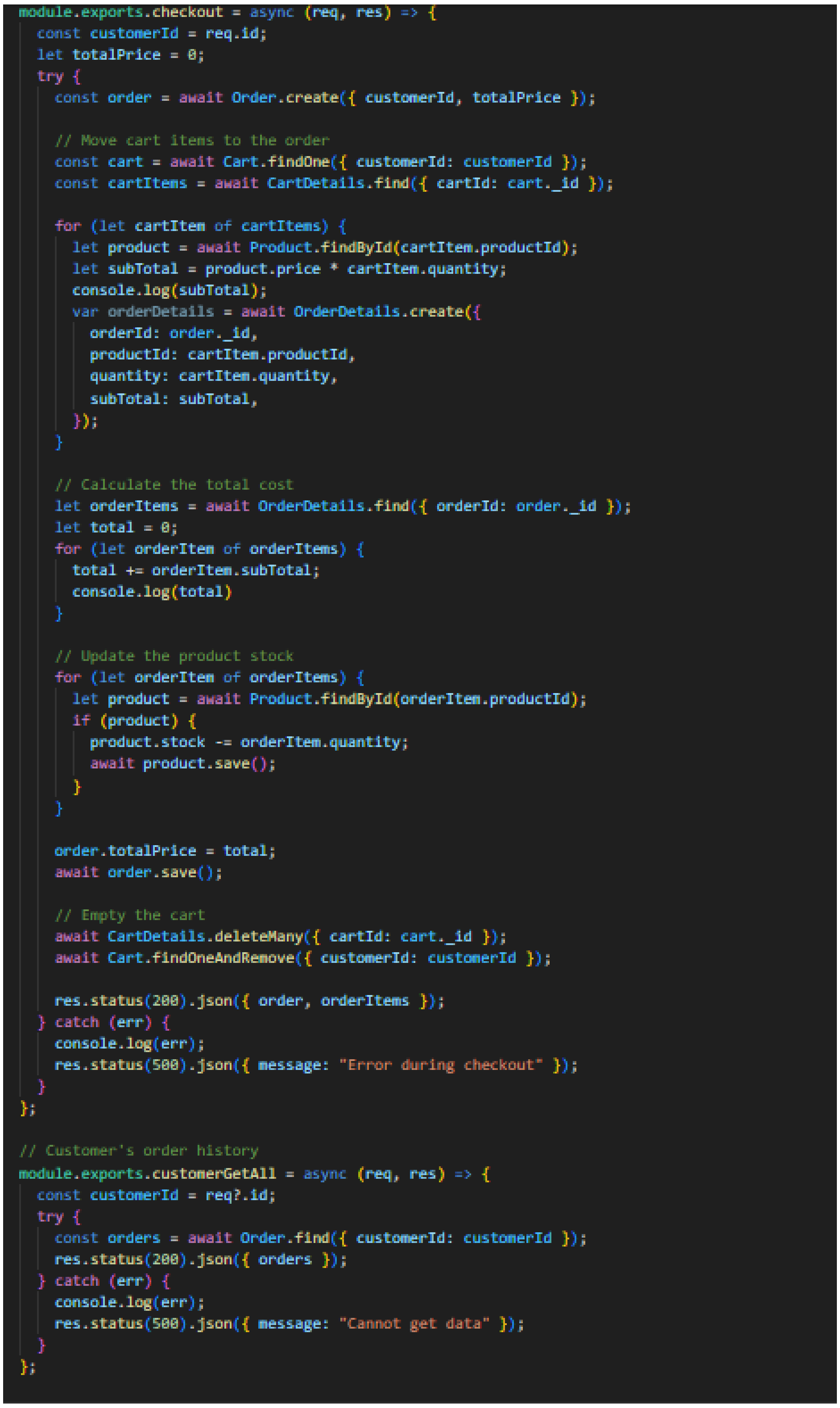
Product:



```
module.exports.getAll = async (req, res) => {
    const page = req?.query.page;
    let category = req?.query?.category;

    if (category) {
        category = category.toLowerCase();
    }

    // Check if there is no page and category specified
    if (page === undefined && category === undefined) {
        try {
            let products = await Product.find().populate("categories", "name");

            // Send products as json
            res.status(200).json({ products });
        } catch (error) {
            res.status(500).json({ message: error });
        }
    } else {
        // page 1 if no page number specified
        const currentPage = req?.query?.page || 1;
        const limit = 12;

        // Count the matching products
        const count = await Product.find(
            category ? { categories: { $in: [category] } } : null
        ).count();
        // Find the maximum page nuber
        const maxPage = Math.ceil(count / limit);
        // Count the products to skip
        const skip = parseInt(page) === 1 ? 0 : page * limit - limit;

        try {
            let products = await Product.find(
                category ? { categories: { $in: { category } } } : null
            )
                .skip(skip)
                .limit(limit)
                .populate("categories", "name");
            res.status(200).json({ products });
        } catch (error) {
            res.status(500).json({ message: err });
        }
    }
};

// Retrieve a product by ID
```

Getting all the products from the database and parsing it as a json so that the ReactJS can handle it and render the product information in the homepage.

Order:



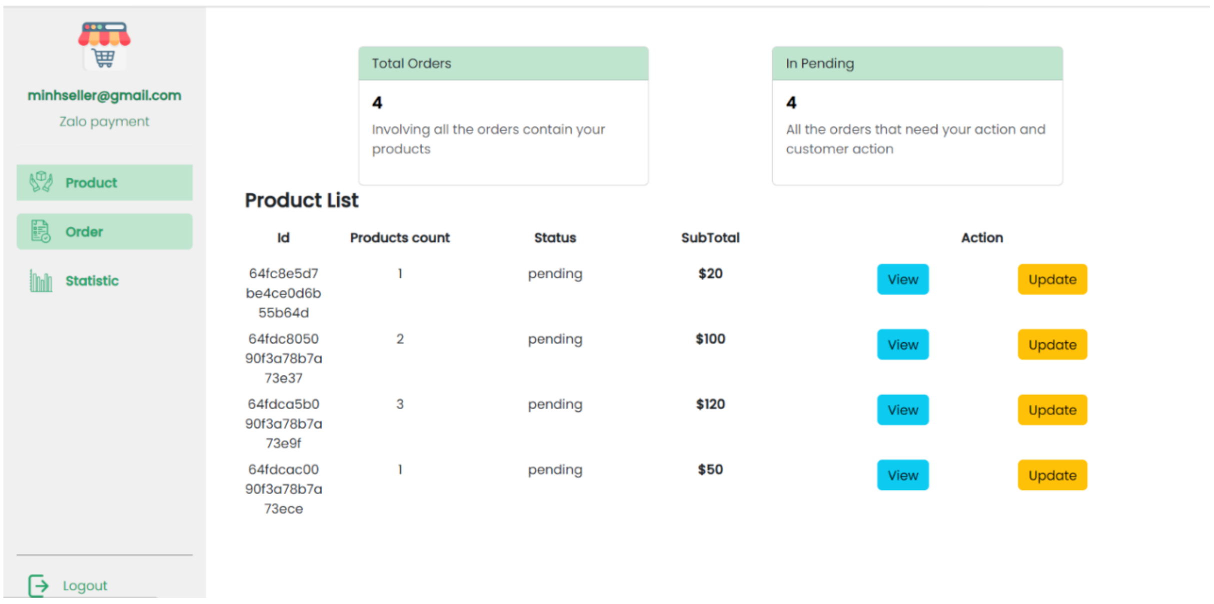
```
module.exports.checkout = async (req, res) => {
  const customerId = req.id;
  let totalPrice = 0;
  try {
    const order = await Order.create({ customerId, totalPrice });

    // Move cart items to the order
    const cart = await Cart.findOne({ customerId: customerId });
    const cartItems = await CartDetails.find({ cartId: cart._id });

    for (let cartItem of cartItems) {
      let product = await Product.findById(cartItem.productId);
      let subTotal = product.price * cartItem.quantity;
      console.log(subTotal);
      var orderDetails = await OrderDetails.create({
        orderId: order._id,
        productId: cartItem.productId,
        quantity: cartItem.quantity,
        subTotal: subTotal,
      });
    }

    // Calculate the total cost
    let orderItems = await OrderDetails.find({ orderId: order._id });
    let total = 0;
    for (let orderItem of orderItems) {
      total += orderItem.subTotal;
      console.log(total)
    }

    // Update the product stock
    for (let orderItem of orderItems) {
      let product = await Product.findById(orderItem.productId);
      if (product) {
        product.stock -= orderItem.quantity;
        await product.save();
      }
    }

    order.totalPrice = total;
    await order.save();

    // Empty the cart
    await CartDetails.deleteMany({ cartId: cart._id });
    await Cart.findOneAndRemove({ customerId: customerId });

    res.status(200).json({ order, orderItems });
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Error during checkout" });
  }
};

// Customer's order history
module.exports.customerGetAll = async (req, res) => {
  const customerId = req?.id;
  try {
    const orders = await Order.find({ customerId: customerId });
    res.status(200).json({ orders });
  } catch (err) {
    console.log(err);
    res.status(500).json({ message: "Cannot get data" });
  }
};
```

The checkout function creates the order in the database. It moves the cart items to the order. Then it calculates the subtotal of all the items in the cart. The total cost is calculated. With the items sold, the product database is then updated with the number of items bought taken away from its stock.

User interface/ User Experience:

The Seller account:

The main page will display the list of products sold by the seller. It will show the total orders containing their products and the orders that are awaiting their approval. You will also get the option to view your products, the stock and update if necessary.



minhseller@gmail.com

Zalo payment

Total Orders

4

Involving all the orders contain your products

In Pending

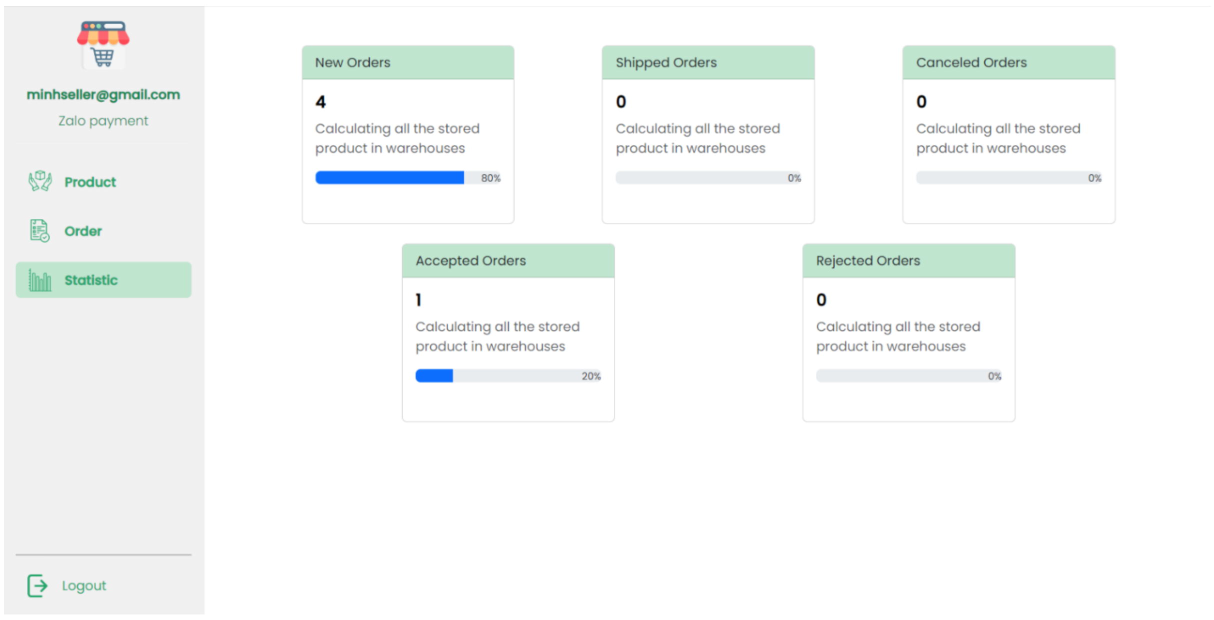
4

All the orders that need your action and customer action

Product List

Id	Products count	Status	SubTotal	Action	
64fc8e5d7be4ce0d6b55b64d	1	pending	$20	View	Update
64fdc805090f3a78b7a73e37	2	pending	$100	View	Update
64fdca5b090f3a78b7a73e9f	3	pending	$120	View	Update
64fdcac0090f3a78b7a73ece	1	pending	$50	View	Update

The Order page shows the orders ordered by the Customer. The Seller has the choice to update the status of the order (Accept, Reject).



minhseller@gmail.com

Zalo payment

New Orders

4

Calculating all the stored product in warehouses

80%

Shipped Orders

0

Calculating all the stored product in warehouses

0%

Canceled Orders

0

Calculating all the stored product in warehouses

0%

Accepted Orders

1

Calculating all the stored product in warehouses

20%

Rejected Orders

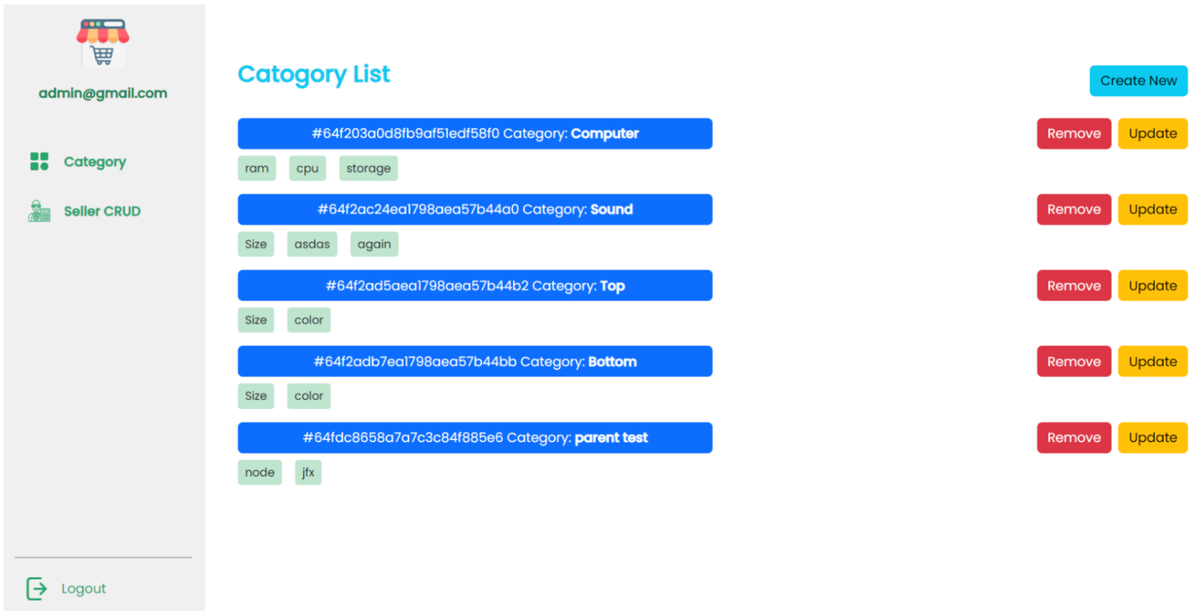
0

Calculating all the stored product in warehouses

0%

The Statistics page shows the statistics of new orders, shipped orders, cancelled orders, accepted orders and rejected orders.

Admin account:



admin@gmail.com

Category

Seller CRUD

Logout

Catogory List

Create New

#64f203a0d8fb9af51edf58f0 Category: **Computer** — Remove Update

ram cpu storage

#64f2ac24ea1798aea57b44a0 Category: **Sound** — Remove Update

Size asdas again

#64f2ad5aea1798aea57b44b2 Category: **Top** — Remove Update

Size color

#64f2adb7ea1798aea57b44bb Category: **Bottom** — Remove Update

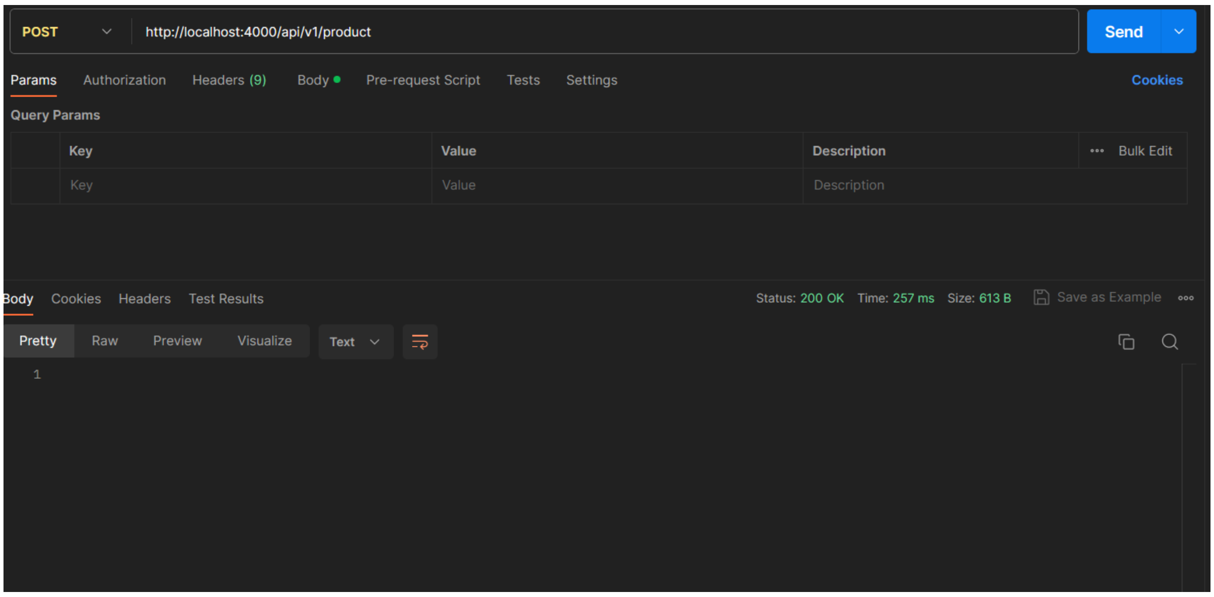
Size color

#64fdc8658a7a7c3c84f885e6 Category: **parent test** — Remove Update

node jfx

The category page shows the category option allowing the administrator to update

Testing:



POST http://localhost:4000/api/v1/product

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers Test Results — Status: 200 OK Time: 257 ms Size: 613 B — Save as Example

Pretty Raw Preview Visualize Text

```
1
```

We test each NodeJS controller using Postman. To test the request and response of each function and check if they have received the data from the

Postman then see what response they have sent back. This is the best way to check how the backend elements would interact with the frontend elements.

2. Development process

The group has a solid workflow, with strict deadlines and meetings. The project is developed on Github. With each member having a project function to work on, each function is then developed on individual branches. After one function is finished, the group members are asked to check upon the work and approve to merge into the main branch. Each week, the group is expected to be present with the meeting in which they will update on the work that they have done, report any problems that arise during the development process.